

Essay:

Construction Theory, Self-Replication, and the Halting Problem

Hiroki Sayama

Department of Bioengineering

Binghamton University, State University of New York

P.O. Box 6000, Binghamton, NY 13902-6000

Tel: 607-777-4439

Fax: 607-777-5780

Email: sayama@binghamton.edu

Abstract

This essay aims to propose *construction theory*, a new domain of theoretical research on machine construction, and use it to shed light on a fundamental relationship between living and computational systems. Specifically, we argue that self-replication of von Neumann's universal constructors holds a close similarity to circular computational processes of universal computers that appear in Turing's original proof of the undecidability of the halting problem. The result indicates the possibility of reinterpreting a self-replicating biological organism as embodying an attempt to solve the halting problem for a *diagonal* input in the context of construction. This attempt will never be completed because of the indefinite cascade of self-computation/construction, which accounts for the undecidability of the halting problem and also agrees well with the fact that life has maintained its reproductive activity for an indefinitely long period of time.

Keywords: von Neumann's universal constructor, construction theory, self-replication, Turing machine, the halting problem

1 Introduction: von Neumann's automata and construction theory

John von Neumann's theory of self-reproducing automata [1, 2] is now regarded as one of the greatest theoretical achievements made in early stages of artificial life research [3, 4, 5]. Before working on its specific implementation on cellular automata, von Neumann sketched a general outline of his self-reproducing automaton that consists of the following parts [1]:

\mathcal{A} : A universal constructor that constructs a product \mathcal{X} from an instruction tape $\mathcal{I}(\mathcal{X})$ that describes how to construct \mathcal{X} .

\mathcal{B} : A tape copier that duplicates $\mathcal{I}(\mathcal{X})$.

\mathcal{C} : A controller that dominates \mathcal{A} and \mathcal{B} and does the following:

1. Give $\mathcal{I}(\mathcal{X})$ to \mathcal{A} and let it construct \mathcal{X} .
2. Pass $\mathcal{I}(\mathcal{X})$ to \mathcal{B} and let it duplicate $\mathcal{I}(\mathcal{X})$.
3. Attach one copy of $\mathcal{I}(\mathcal{X})$ to \mathcal{X} and separate $\mathcal{X} + \mathcal{I}(\mathcal{X})$ from the rest.

The functions of these parts are symbolically written as

$$\mathcal{A} + \mathcal{I}(\mathcal{X}) \rightarrow \mathcal{A} + \mathcal{I}(\mathcal{X}) + \mathcal{X}, \tag{1}$$

$$\mathcal{B} + \mathcal{I}(\mathcal{X}) \rightarrow \mathcal{B} + 2\mathcal{I}(\mathcal{X}), \tag{2}$$

$$\begin{aligned} & (\mathcal{A} + \mathcal{B} + \mathcal{C}) + \mathcal{I}(\mathcal{X}) \\ & \rightarrow ((\mathcal{A} + \mathcal{I}(\mathcal{X})) + \mathcal{B} + \mathcal{C}) \\ & \rightarrow ((\mathcal{A} + \mathcal{I}(\mathcal{X}) + \mathcal{X}) + \mathcal{B} + \mathcal{C}) \\ & \rightarrow (\mathcal{A} + (\mathcal{B} + \mathcal{I}(\mathcal{X})) + \mathcal{C}) + \mathcal{X} \\ & \rightarrow (\mathcal{A} + (\mathcal{B} + 2\mathcal{I}(\mathcal{X})) + \mathcal{C}) + \mathcal{X} \\ & \rightarrow (\mathcal{A} + \mathcal{B} + \mathcal{C}) + \mathcal{I}(\mathcal{X}) + \mathcal{X} + \mathcal{I}(\mathcal{X}) \\ & \rightarrow \{(\mathcal{A} + \mathcal{B} + \mathcal{C}) + \mathcal{I}(\mathcal{X})\} + \{\mathcal{X} + \mathcal{I}(\mathcal{X})\}. \end{aligned} \tag{3}$$

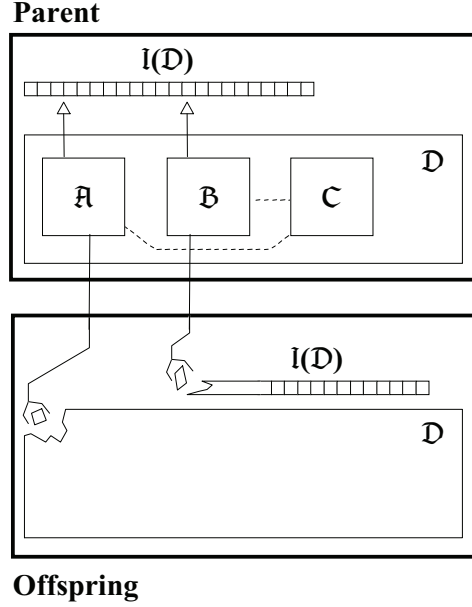


Figure 1: Schematic illustration of von Neumann’s self-reproducing automaton. It consists of three parts (universal constructor \mathcal{A} , tape copier \mathcal{B} , and controller \mathcal{C} ; $\mathcal{D} \equiv \mathcal{A} + \mathcal{B} + \mathcal{C}$) and an instruction tape $\mathcal{I}(\mathcal{D})$. Since $\mathcal{I}(\mathcal{D})$ contains the information about how to construct the automaton \mathcal{D} itself, the whole system $\mathcal{D} + \mathcal{I}(\mathcal{D})$ can self-replicate.

Then self-replication can be achieved if one lets $\mathcal{X} = \mathcal{D} \equiv \mathcal{A} + \mathcal{B} + \mathcal{C}$, i.e.,

$$\mathcal{D} + \mathcal{I}(\mathcal{D}) \rightarrow \{\mathcal{D} + \mathcal{I}(\mathcal{D})\} + \{\mathcal{D} + \mathcal{I}(\mathcal{D})\}. \quad (4)$$

Figure 1 illustrates these notations visually. Note that for the above conclusion to apply, \mathcal{D} must be within the product set of \mathcal{A} , which is by no means trivial.

Alan Turing’s preceding work on computationally universal machines [6] gave a hint for von Neumann to develop these formulations of self-reproducing automata, especially on the idea of universal constructor \mathcal{A} . These two kinds of machines apparently share a similar concept that a universal machine, given an appropriate finite description, can execute arbitrary tasks specified in the description. We should note however, that this similarity has often been overstated in the literature, leading to some misunderstandings of von Neumann’s original intention, recently argued by McMullin [5, 7]. The most significant difference between these two types of universal machines is that the constructional machine must be made of the same

parts that it operates on, and therefore both the machine and the parts must be embedded in the same space-time and obey the same “physical” rules, while the computational machine can be separate from the symbols it operates on, like the Turing machine’s head that exists outside its tape. Another equally important difference is that computational universality is defined by the ability of *computing the behavior of all the other models of computation*, while the constructional universality is defined by the ability of *constructing all the structures in a given specific product set*, which has nothing to do with the ability of computing the behavior of other constructors. The latter issue will be revisited later.

The aforementioned differences suggest the need for a distinct domain of research specially dedicated to the issues of machine construction, pioneered by von Neumann’s work on constructional machines but since left unnamed to date. This would be closely related to computation theory pioneered by Turing’s work, but should be unique by involving physical interpretation and implementation of production processes and thereby connecting logic and mathematics to biology and engineering. Here I propose to call it *construction theory*, a domain of research that focuses on the theoretical aspects of productive behaviors of natural or artificial systems, including self-replication, self-repair and other epigenetic processes. There is a recent resurgence of studies on these topics in artificial life and other related fields [8, 9, 10, 11, 12, 13]. Like in computation theory, there are many important problems yet to be addressed in construction theory, such as identifying the class of constructible structures with a given set of physical rules; obtaining necessary/sufficient conditions for a universal constructor to exist for a given product set; determining whether there is a single *truly universal* construction model that could emulate all other construction models; etc.

In what follows, we will focus on one particular question regarding the relationship between computation and construction theories. While von Neumann’s universal constructor was largely inspired by Turing’s universal computer, what the entire self-replicating automaton \mathcal{D} in construction theory would parallel in computation theory remained unclear to many, perhaps because von Neumann himself did not detail in his writings how his theoretical model was related to computation theory. Besides the universal constructor \mathcal{A} , the automaton \mathcal{D} also includes \mathcal{B} that duplicates a given tape and \mathcal{C} that attaches a copy of the duplicated tapes to the product of \mathcal{A} . They are the subsystems that von Neumann added to the automaton in view of self-replication (and subsequent evolutionary processes). Their counterparts are not present in the design

of Turing machines, and therefore, the entire architecture of self-reproducing automata has often been considered a heuristic design meaningful only on the construction side, but not on the computation side.

Here I would like to help readers realize that self-replication in construction theory actually has a fundamental relationship with the diagonalization proof of the undecidability of the halting problem in computation theory. This relationship was already suggested by some mathematicians and theoretical computer scientists [14, 15]; however, it somehow failed to bring a broader conceptual impact to other related fields, including theoretical biology, artificial life, and complex systems research. Specifically, the mathematical description of self-replication by von Neumann’s universal constructors is of identical form with the circular computational processes of universal computers that appear in Turing’s original proof of the undecidability of the halting problem. This leads us to a new interpretation of a self-replicating biological organism as embodying an attempt to solve the undecidable halting problem for a *diagonal* input, not in computation theory but in the context of von Neumann’s construction theory. This attempt, of course, will never be completed in a finite time because of the indefinite cascade of self-computation/construction, which accounts for the undecidability of the halting problem and also agrees well with the fact that life has maintained its reproductive activity for an indefinitely long period of time.

2 The halting problem

The halting problem is a well-known decision problem in theoretical computer science that can be informally described as follows:

Given a description of a computer program and an initial input it receives, determine whether the program eventually finishes computation and halts on that input.

This problem has been one of the most profound issues in computation theory since 1936 when Turing proved that no general algorithm exists to solve this problem for any arbitrary programs and inputs [6]. The conclusion is often paraphrased that the halting problem is *undecidable*.

Turing’s proof uses *reductio ad absurdum*. A well-known simplified version takes the following three steps.

First, assume that there is a general algorithm that can solve the halting problem for any program p and input i . This means that there must be a Turing machine M that always halts for any p and i and computes the function

$$f(p, i) \equiv \begin{cases} 1 & \text{if the program } p \text{ halts on the input } i, \\ 0 & \text{otherwise.} \end{cases} \quad (5)$$

Second, one can easily derive from this machine another Turing machine M' whose behavior is modified to compute only *diagonal* components in the p - i space, i.e.,

$$f'(p) \equiv f(p, p). \quad (6)$$

This machine determines whether the program p halts when its self-description is given to it as an input. Such self-reference would be meaningless for most actual computer programs, but still theoretically possible.

Then, finally, one can tweak M' slightly to make yet another machine M^* that falls into an infinite loop if $f'(p) = 1$. What could happen if M^* was supplied with its self-description $p(M^*)$? It eventually halts if $f'(p(M^*)) = f(p(M^*), p(M^*)) = 0$, i.e., if it does not halt on $p(M^*)$. Or, it loops forever if $f'(p(M^*)) = f(p(M^*), p(M^*)) = 1$, i.e., if it eventually halts on $p(M^*)$. Both lead to contradiction. Therefore, the assumption we initially made must be wrong—there must be no general algorithm to solve the halting problem.

3 Turing's original proof

Here I would like to bring up an informative yet relatively untold fact that Turing himself did not like to have such a tricky mathematical treatment as the above third step that introduces a factitious logical inversion into the mechanism of the machine, so he intentionally avoided using it in his original proof. Below is a quote from his original paper [6, p.246], which tells us how unique Turing's thought was and how much emphasis he placed on an intuitive understanding of mathematical concepts:

“... The simplest and most direct proof of [the fact that there is no general process for determining whether a given program continues to write symbols indefinitely] is by showing that, if this general

process exists, then there is a machine which computes J^1 . This proof, although perfectly sound, has the disadvantage that it may leave the reader with a feeling that “there must be something wrong”. The proof which I shall give has not this disadvantage, ...”

(Footnote added by the author)

Instead, what he actually did for the proof was summarized in the following paragraph [6, p.247]:

“... Now let K be the $D.N^2$ of H^3 . What does H do in the K -th section of its motion? It must test whether K is satisfactory⁴, giving a verdict “s” or “u”. Since K is the $D.N$ of H and since H is circle-free, the verdict cannot be “u”. On the other hand the verdict cannot be “s”. For if it were, then in the K -th section of its motion H would be bound to compute the first $R(K - 1) + 1 = R(K)^5$ figures of the sequence computed by the machine with K as its $D.N$ and to write down the $R(K)$ -th as a figure of the sequence computed by H . The computation of the first $R(K) - 1$ figures would be carried out all right, but the instructions for calculating the $R(K)$ -th would amount to “calculate the first $R(K)$ figures computed by H and write down the $R(K)$ -th”. This $R(K)$ -th figure would never be found. I.e., H is circular, contrary both to what we have found in the last paragraph and to the verdict “s”. Thus both verdicts are impossible and we conclude that there can be no machine D^6 .”

(Footnotes added by the author)

In this paragraph, Turing considered the *actual behavior* of intact H on its self-description K , and noticed that what this machine would need to compute is exactly the same situation as the machine itself is in:

¹A binary sequence whose n -th digit is a Boolean *inverse* of the n -th digit of the n -th computable sequence. If this sequence is computable, then it must be listed somewhere in the series of the computable sequences, which however causes a contradiction because its diagonal element must be both 0 and 1 at the same time. Therefore this sequence cannot be computable.

²Description Number: An integer that describes the specifics of a given computational machine.

³A machine that incrementally and indefinitely computes the diagonal sequence of the infinite matrix made of all the infinitely long computable sequences enumerated in the order of D.N's of corresponding machines.

⁴An integer N is considered satisfactory if the machine whose D.N is N can keep writing symbols indefinitely without falling into a deadlock (Turing called this property *circle-free*).

⁵ $R(N)$ denotes how many machines are circle-free within the range of D.N's up to N .

⁶A machine that is assumed capable of determining whether a given machine is circular or not. This machine is introduced to construct H .

“*H is looking at its self-description K.*” Such a self-reference would result in a circular process that never comes back. Therefore, H cannot make any decision on whether K is satisfactory or not. This contradiction negatively proves the possibility of D , or a general computational procedure to determine whether a machine stops writing symbols or not.

4 Self-replication emerging

Turing’s argument described above gives essentially the same argument as to what could happen if M' in our notation received its self-description $p(M')$. In this case M' must compute the value of $f'(p(M')) = f(p(M'), p(M'))$, and hence it would need to compute the behavior of the machine described in $p(M')$ on the input $p(M')$, exactly the same circular situation as that appearing in Turing’s proof. Let us use this example in what follows, as it is much simpler to understand than Turing’s original settings.

What kind of computational task would M' be carrying out in this circular situation? It tries to compute the behavior of $M' + p(M')$, which tries to compute the behavior of $M' + p(M')$, which tries to compute the behavior of $M' + p(M')$, ... Interestingly, this chain of self-computation takes place in the form identical to that of self-replication in von Neumann’s construction theory shown in Eq. (4), if “*to compute the behavior*” is read as “*to construct the structure*”. This similarity may be better understood by noting that the role of \mathcal{C} that attaches $\mathcal{I}(\mathcal{X})$ to \mathcal{X} , shown in the last line of Eq. (3), parallels the role of diagonalization in Eq. (6); *both attempt to apply a copy of the description to the machine represented by the description.*

Moreover, if one watched how the actual configuration of the tape of M' changes during such a self-computing process, he would see that the information about M' *actually self-replicates* on the tape space, with its representation becoming more and more indirect as the level of self-computation becomes deeper (Fig. 2). Turing might have imagined this kind of self-replicating dynamics of machines when he developed his argument.

In view of the similarity between the above two processes, it is clearly recognized that von Neumann’s design of self-reproducing automata is by no means just an anomaly in construction theory. Rather, it correctly reflects the diagonal situation leading to an infinite self-computation chain of computationally universal machines, which appears in the proof of the undecidability of the halting problem presented by

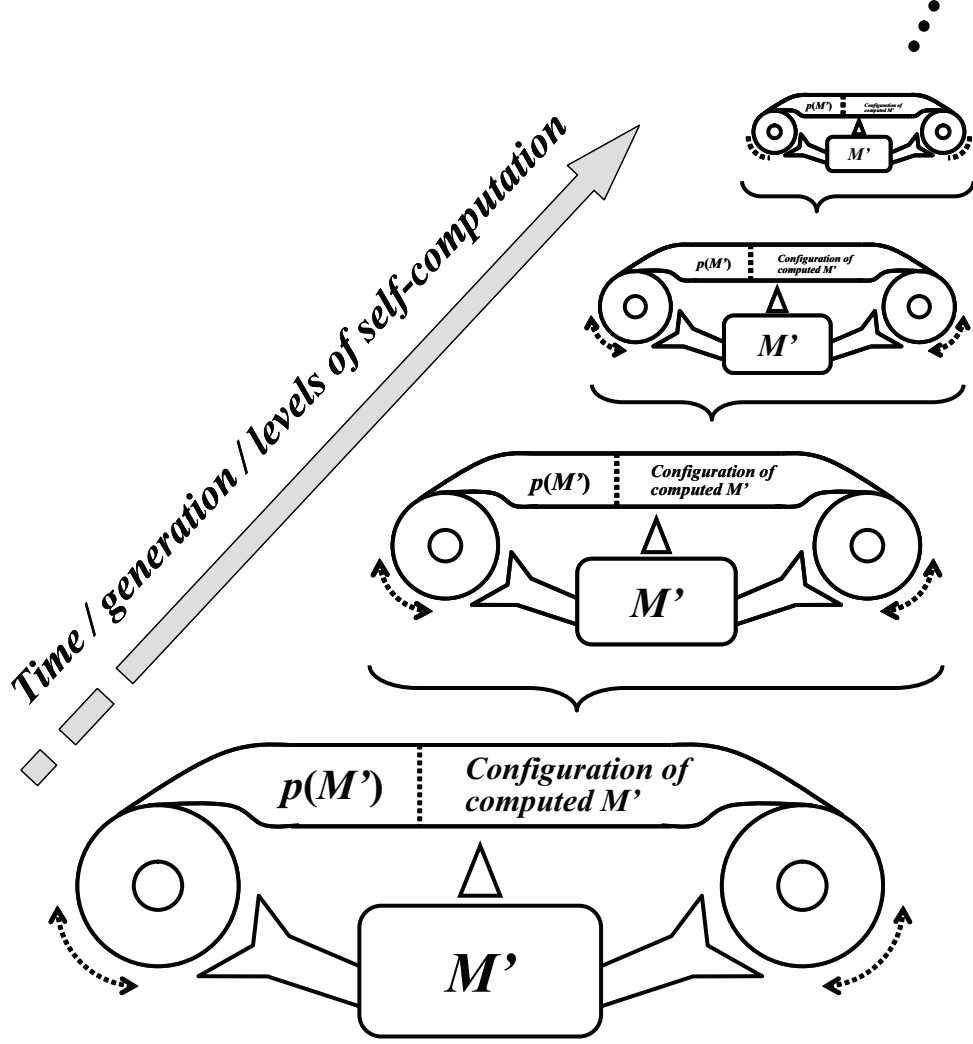


Figure 2: Self-replication of Turing machine M' on the tape space. Given its own description $p(M')$, it starts an indefinite cascade of self-computation, where the information about M' actually self-replicates on the tape. The representation of the computed machine becomes more and more indirect as the level of self-computation becomes deeper.

Turing.

5 Related work

A well-known argument on the computational undecidability related to self-replication was developed by Cohen [16], where he showed that there is no general algorithm for the detection of self-replicating computer viruses. The proof is rather simple: If there were an algorithm, say S , that can determine whether a given computer program is self-replicative, then one could easily create another contradictory program that has S built in it and self-replicates if and only if its S classifies the program itself as non-self-replicative. This is probably the best acknowledged discussion on the relationship between self-replication and the undecidable problem so far.

We should note, however, that Cohen's argument suggests that detecting a computer program that does "X" is generally impossible, where "X" could be self-replication but could also be replaced by any other functions; self-replication is no more than just one of many possible behaviors of universal machines. In contrast, our argument discussed in this essay is more fundamental: Universal machines may fall into undecidable situations *because of the possibility of self-replication (either computation or construction)*. Here self-replication is not just an instance of many possible behaviors, but is actually the key property that causes the undecidability of the behavior of universal machines, either computational or constructional.

Another related work would be the theory of self-replicative recursive functions discussed in recursion theory in 1960's, where Kleene's recursion theorem [17] was applied to prove that there exist recursive functions (i.e., computer programs) that produce their own representations as outputs, regardless of given inputs [14, 15]. These functions were later implemented as actual computer programs and named *quines* [18]; writing quines in various programming languages has been one of the standard amusements in computer science community. A common way of creating a quine is to embed a partial representation of the program in itself and use it twice for creating a main body of the program and for re-embedding a copy of the representation into the newly created program. This technique of *quining* is exactly the same as what von Neumann proposed in his formulation of self-replicating machines.

There is, however, at least one fundamental difference between these self-replicating programs in recursion

theory and the self-replicating constructors in construction theory. In the former case, the computation process always stops after producing a static representation of the program, with no direct implication derived on its relevance to the halting problem. In the latter case, on the other hand, the construction process never stops because the product of construction, as active as its constructor, starts its own construction process once fully constructed.

Interestingly, Turing’s argument in his proof of the undecidability of the halting problem considered the *latter* case in computation theory, where the Turing machine is not simply writing its own representation (as usual quine programs do), but is actually trying to compute its own dynamic behavior (as illustrated in Fig. 2). This point may be well understood by reminding that a product being constructed in construction theory corresponds not to symbols being written, but to a *computational process itself*, in computation theory.

6 Conclusion

As Turing showed in his proof, when a computational machine tries to solve the halting problem of its own computation process, it will fall into a cycle of self-computation that never ends in a finite time. Our point is that this corresponds exactly to the cycle of self-replication in construction theory, and that von Neumann’s self-reproducing automaton model rightly captures this feature in its formulation. The halting problem solver in construction theory lets the subject machine construct its product and see if it eventually stops. If it tries to solve the halting problem of its own construction process, it will start self-replication, and the entire process never completes in a finite amount of time.

The insight obtained in the above sections provides us with some new implications about the connections between computation and construction. Throughout our argument, we saw that the construction of another machine in construction theory has the same role and meaning as does the computation of another machine in computation theory. This correspondence transcends the second difference between computational and constructional machines we discussed in the Introduction, where I said:

The computational universality is defined by the ability of computing the behavior of all the other models of computation, while the constructional universality is defined by the ability of constructing all the structures in a given specific product set, which has nothing to do with the

ability of computing the behavior of other constructors.

Interestingly, once construction and computation are identified with each other, these two universalities become very close—if not exactly the same—so that the universal constructor indeed has the ability to compute the behavior of all the other constructors *by physically constructing them and letting them do their jobs*. The idea of such “constructor-constructors” is relevant to the realization of machines with epigenetic dynamics, which will be one of the more important subjects in construction theory.

The computation-construction correspondence also gives us a unique view of biology, suggesting that the relationship between parent and offspring in biological systems is equivalent to the relationship between the *computing* M' and the *computed* M' in computation theory. From a construction-theoretic perspective, a biological organism is trying to find out the final result of the construction task written in its genotypic information by executing its contents. The final product will be immediately found if the product of the task is a static structure, such as drugs produced by genetically modified bacteria. But if the product is another active machine that will attempt to build other products, then the final result will depend on what this product will do next. Furthermore, if the product is identical (or sufficiently similar) to the original organism itself, the situation represents the conventional parent-offspring relationship, where offspring are a kind of intermediate product produced during the whole long-standing construction process.

In this view, the endless chain of self-replication that living systems are in, may be reinterpreted as a parallel to the endless chain of self-computation that a halting problem solver falls in. In a sense, we may all be in the process initiated billions of years ago by a first universal constructor, who just tried to see the final product of its *diagonal* construction.

Acknowledgments

I would like to thank William R. Buckley for his continuous encouragement and helpful suggestions, and also four anonymous reviewers who gave me very constructive and insightful comments that significantly improved the quality and correctness of the ideas presented here.

References

- [1] von Neumann, J. (1951). The general and logical theory of automata. In Jeffress, L. A., ed., *Cerebral Mechanisms in Behavior—The Hixon Symposium*, pp.1–41. New York: John Wiley. Originally presented in September, 1948. Also collected in Aspray, W., and Burks, A. W., eds., *Papers of John von Neumann on Computing and Computer Theory*, pp.391–431. 1987. Cambridge, MA: MIT Press.
- [2] von Neumann, J. (1966). *Theory of Self-Reproducing Automata*. Urbana, IL: University of Illinois Press. Edited and completed by A. W. Burks.
- [3] Marchal, P. (1998). John von Neumann: The founding father of artificial life. *Artificial Life*, 4, pp.229–235.
- [4] Sipper, M. (1998). Fifty years of research on self-replication: An overview. *Artificial Life*, 4, pp.237–257.
- [5] McMullin, B. (2000). John von Neumann and the evolutionary growth of complexity: Looking backward, looking forward... *Artificial Life*, 6, pp.347–361.
- [6] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc. Ser. 2*, 42, pp.230–265. A correction followed in 1937. Fulltext available online at <http://www.abelard.org/turpap2/tp2-ie.asp>.
- [7] McMullin, B. (1993). What is a universal constructor? *Dublin City University School of Electronic Engineering Technical Report bmcm9301*.
- [8] Freitas Jr., R. A. & Merkle, R. C. (2004). *Kinematic Self-Replicating Machines*. Georgetown, TX: Landes Bioscience.
- [9] Zykov, V., Mytilinaios, E., Adams, B. & Lipson, H. (2005). Self-reproducing machines. *Nature*, 435, pp.163–164.
- [10] Buckley, W. R. & Bowyer, A. eds. (2006). Workshop on Machine Self-Replication. In *Workshop Proceedings of the Tenth International Conference on the Simulation and Synthesis of Living Systems (ALIFE X)*. pp.125–151. Fulltext available online at <http://www.alifex.org/program/wkshp-proceed.pdf>.

- [11] Ewaschuk, R. & Turney, P. D. (2006). Self-replication and self-assembly for manufacturing. *Artificial Life*, 12, pp.411–433.
- [12] Suzuki, K. & Ikegami, T. (2006). Spatial-pattern-induced evolution of a self-replicating loop network. *Artificial Life*, 12, pp.461–485.
- [13] Zhang, X., Dragffy, G. & Pipe, A. G. (2006). Embryonics: A path to artificial life? *Artificial Life*, 12, pp.313–332.
- [14] Myhill, J. (1964) The abstract theory of self-reproduction. In Mesarović, M. D., ed., *Views on General Systems Theory: Proceedings of the Second Systems Symposium at Case Institute of Technology*, pp.106–118. John Wiley & Sons.
- [15] Rogers, H. Jr. (1967) *Theory of Recursive Functions and Effective Computability*, pp.188–190. McGraw-Hill.
- [16] Cohen, F. (1987). Computer viruses: Theory and experiments. *Computers & Security*, 6, pp.22–35.
- [17] Kleene, S. (1938) On notation for ordinal numbers. *Journal of Symbolic Logic* 3:150–155.
- [18] Hofstadter, D. (1979) *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books.